Effective C++

항목 1: C++를 언어들의 연합체로 바라보는 안목은 필수

C++를 가장 쉽게 이해하는 방법

-> C++를 단일 언어로 바라보는 눈을 넓혀, 상관관계가 있는 여러 언어들의 연합체로 보기

C++의 하위 언어를 알아야 제대로 C++를 이해했다고 할 수 있다.

1. C 언어

블록, 문장, 선행 처리자, 기본제공 데이터타입, 배열, 포인터 등 모두 C에서 왔다. 하지만 템플릿, 예외, 오버로딩 등이 없다.

1. 객체 지향 개념의 C++

클래스(생성자&소멸자), 캡슐화, 상속, 다형성, 가상 함수 등

1. 템플릿 C++

새로운 프로그래밍 패러다임이 파생됨 -> 템플릿 메타프로그래밍(TMP)

1. STL

컨테이너, 반복자, 알고리즘과 함수 객체가 얽혀 돌아가는 규약

네 가지 하위 언어들이 C++를 이루고 있다. 효과적인 프로그램 개발을 위해 하위 언어들의 대응 전략을 알아야 합니다. 따라서 C++는 네 가지 하위 언어들의 연합체이며 각각의 하위 언어가 자신만의 규칙을 가지고 있습니다.

C++를 사용한 효과적인 프로그래밍 규칙은 어떤 하위 언어를 사용하느냐에 따라 달라집니다.

항목 2: #define을 쓰려거든 const, enum, inline을 떠올리자

컴파일러 > 선행 처리자

예) #define ASPECT\_RATIO 1.653 은 컴파일러에겐 전혀 기호식 이름으로 보이지 않는다.

이유) 소스코드가 어떻게든 컴파일러에게 넘어가기 전에 선행 처리자가 밀어버리고 숫자 상수로 바꾸어 버리기 때문 그 결과 ASPECT\_RATIO라는 이름은 컴파일러가 쓰는 기호 테이블에 들어가지 않는다. 컴파일 에러가 발생하면 소스 코드에는 ASPECT\_RATIO가 있었는데 에러 메시지엔 1.653이 있어 이게 어디서 왔는지 모르고 찾느랴 시간을 허비할 수도 있다.

해결법 -> 매크로 대신 상수를 쓰는 것. Const double AspectRatio = 1.653;

이때 좋은 점. 컴파일러의 눈에 보임 / 기호 테이블에 들어감 / 컴파일을 거친 최종 코드의 크기가 #define을 썼을 때보다 작게 나올 수 있다. 이유는 ASPECT\_RATIO가 등장하기만 하면 선행 처리자에 의해 1.653의 사본이 등장 횟수만큼 들어가게 됩니다.

주의 사항 2가지

첫째 상수 포인터를 정의하는 경우. 상수 정의는 헤더 파일에 넣는 것이 상례(그래야 다른 소스 파일이 이것을 인클루드해서 씀) 포인터는 꼭 const로 선언해 주고 포인터가 가리키는 대상까지 const로 선언.

예) const char \* const authorName = “Scott Meyers”;

예) const std::string authorName(“Scott Meyers”);

둘째 클래스 멤버로 상수를 정의하는 경우, 즉 클래스 상수를 정의하는 경우

선언 vs 정의 (선언이나 정의 모두 값 할당과는 무관)

선언 -> 컴파일러에게 사용될 요소를 알려주는 것(메모리 주소 바인딩X)

정의 -> 코드나 데이터 같은 요소를 구체적으로 명시한 것.(메모리 주소 바인딩O)

Class GamePlayer

{

Private:

Static cons tint NumTurns = 5; // 상수 선언

Int scores[NumTurns]; // 상수를 사용하는 부분

};

C++에서는 사용하고자 하는 것에 대해 ‘정의’가 마련되어 있어야 하는 게 보통인데 정적 멤버로 만들어지는 정수류(정수, char, bool 등) 타입의 클래스 내부 상수는 예외 이다.

이때 정의를 제공하고자 할때는 const int GamePlayer::NumTurns;(클래스 상수의 정의) 를 구현 파일에 둡니다.

클래스 상수의 초기값은 해당 상수가 선언된 시점에서 바로 주어지기 때문에 정의에는 상수의 초기값이 있으면 안 됨.

만약 위 문법을 받아 들이지 않는 오래된 컴파일러를 사용 할 때는 초기값을 상수 ‘정의’ 시점에 주도록 함.

Class CostEstimate{

Private:

Static const double FudgeFactor; // 정적 클래스 상수의 선언

// 이것은 헤더 파일에 둡니다.

};

Const double // 정적 클래스 상수의 정의

CostEstimate::FudgeFactor = 1.35; // 이것은 구현 파일에 둡니다.

클래스를 컴파일 하는 도중에 클래스 상수의 값이 필요할 때

Class GamePlayer{

Private:

Enum { NumTurns = 5 }; // “나열자 둔갑술”: NumTurns를 5에 대한 기호식 이름으로 만듭니다.

Int scores[NumTurns]; // 해결!

};

나열자 둔갑술 좋은점!

나열자 둔갑술의 동작 방식은 const보다 #define에 가깝다.

선언한 정수 상수를 가지고 다른 사람이 주소를 얻는다든지 참조자를 쓴다든지 하는 것이 싫다면 enum이 좋은 자물쇠가 될 수 있습니다. Enum은 #define처럼 어떤 형태의 쓸데없는 메모리 할당도 절대 저지르지 않습니다.

지극히 실용적이다. 템플릿 메타프로그래밍의 핵심 기법이기도 합니다.

#define 지시자의 또 다른 오용 사례는 매크로 함수입니다.

// a와 b 중에 큰 것을 f에 넘겨 호출 합니다.

#define CALL\_WITH\_MAX(a, b) f((a) > (b) ? (a) : (b)) // 안좋은 예 입니다.

이유 괴현상 발생

예) int a = 5, b = 0;

CALL\_WITH\_MAX(++a, b); // a가 두 번 증가합니다.

CALL\_WITH\_MAX(++a, b+10); // a가 한 번 증가합니다.

위 문제 해결 방법

인라인 함수에 대한 템플릿을 준비하는 것.

Template<typename T>//T가 정확히 무엇인지 모르기 때문에, 매개변수로 상수 객체에 대한 참조자를 씁니다. 항목 20 참조

Inline void callWithMax(const T& a, const T& b)

{

f(a > b ? a : b);

}

template<typename T>//T가 정확히 무엇인지 모르기 때문에, 매개변수로 상수 객체에 대한 참조자를 씁니다. 항목 20 참조

inline int callWithMax(const T& a, const T& b)

{

int f = (\*a > \*b ? \*a : \*b);

return f;

}

void main()

{

int a = 10;

int b = 11;

int c = callWithMax(&a, &b);

cout << "c = " << c << endl;

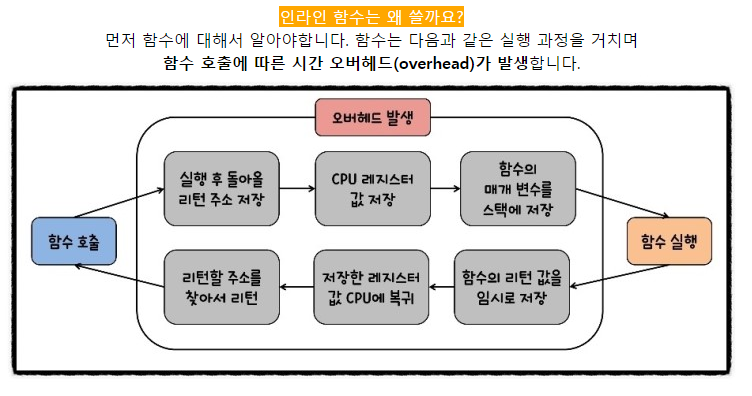
}

이 함수는 템플릿이기 때문에 동일 계열 함수군을 만들어 냅니다. 동일한 타입의 객체 두 개를 인자로 받고 둘 중 큰 것을 f에 넘겨서 호출하는 구조 입니다. 그리고 진짜 함수이기 때문에 유효범위 및 접근 규칙을 그대로 따라갑니다.

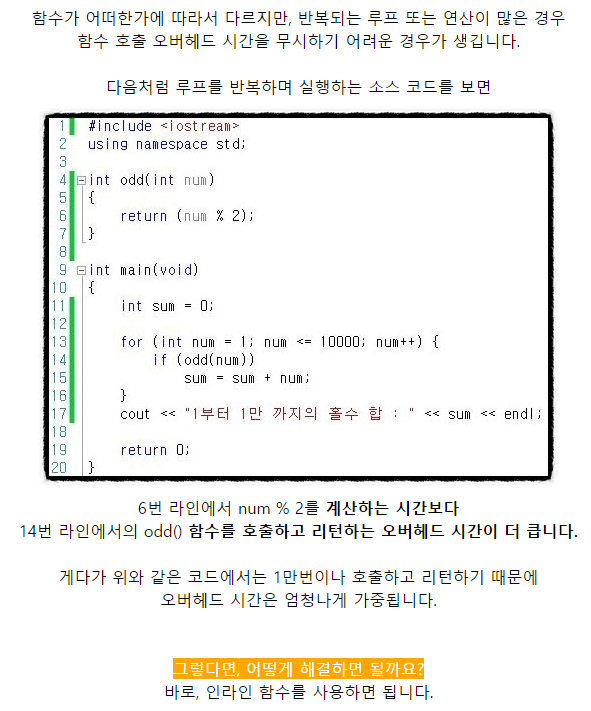
이것만은 잊지 말자!

* 단순한 상수를 쓸 때는, #define보다 const 객체 혹은 enum을 우선 생각합니다.
* 함수처럼 쓰이는 매크로를 만들려면, #define 매크로보다 인라인 함수를 우선 생각합시다.

인라인 함수 설명 <http://m.blog.naver.com/madplay/220236764801>



오버헤드를 줄일 수 있다.



항목 3: 낌새만 보이면 const를 들이대 보자!

‘의미적인 제약’(const 키워드가 붙은 객체는 외부 변경을 불가능하게 한다)을 소스 코드 수준에서 붙인다는 점과 컴파일러가 이 제약을 단단히 지켜준다.

어떤 값이 불변이어야 한다는 제작자의 의도를 컴파일러 및 다른 프로그래머와 나눌 수 있는 수단.

클래스 바깥에서는 전역 혹은 네임스페이스 유효범위의 상수를 선언하는 데 씀

클래스 내부에서는 정적 멤버 및 비정적 데이터 멤버 모두를 상수로 선언할 수 있습니다.

기본적으로는 포인터 자체를 상수로, 혹은 포인터가 가리키는 데이터를 상수로 지정할 수 있는데, 둘 다 지정할 수도 있고 아무것도 지정하지 않을 수도 있다.

char greeting[] = “Hello”;

const char \* p = greeting;// 포인터가 가리키는 대상이 상수(상수 데이터)

char \* const p = greeting;// 포인터 자체가 상수(상수 포인터)

즉, \* 왼쪽에 const가 있으면 가리키는 대상이 상수 이고, 오른쪽에 const가 있으면 포인터 자체가 상수 이다.

어떤 반복자를 const로 선언하는 일은 포인터를 상수로 선언하는 것과 같다. 반복자는 자신이 가리킬 수 있는 것만 가리 킬 수 있고 가리킬 수 있는 대상 자체는 변경이 가능 합니다.

만약 변경이 불가능한 객체를 가리키는 반복자가 필요하다면 const\_iterator를 쓰면 됩니다.

const std::vector<int>::iterator iter = vec.begin();// iter는 T\* const처럼 동작(상수 포인터)

(\*iter) = 10; // 가능 iter가 가리키는 대상을 변경

++iter; // 에러! iter는 상수 입니다.

std::vector<int>::const\_iterator cIter = vec.begin(); // cIter는 const T\*처럼 동작합니다.(상수 데이터)

(\*cIter) = 10; // 에러! (\*cIter)가 상수이기 때문에 안 됩니다.

++cIter; // 가능 포인터(cIter)는 상수가 아니기 때문입니다.

함수 반환 값을 상수로 정해 주면, 안전성이나 효율을 포기하지 않고도 사용자측의 에러 돌발 상황을 줄이는 효과를 볼 수 있다.

예) class Rational{ … };

const Rational operator\*(const Rational& lhs, const Rational& rhs);

이렇게 반환 값이 상수 객체일 경우 아래와 같이 어처구니 없는 실수를 막을 수 잇습니다.

Rational a, b, c;

(a \* b) = c; // a \* b의 결과에 operator=를 호출하다니!

if ( a \* b = c ) .. // 원래 목적은 a\*b와 c를 비교하려는 건데!

매개 변수 혹은 지역 객체를 수정할 수 없게 하는 것이 목적이라면 const로 선언하는 것을 잊지 말도록 합시다.

상수 멤버 함수

멤버 함수에 붙는 const는 “해당 멤버 함수가 상수 객체에 대해 호출될 함수이다”

이렇게 쓰는 이유

첫째. 클래스의 인터페이스를 이해하기 좋게 하기 위해

둘째. 이 키워드를 통해 상수 객체를 사용할 수 있게 함

C++의 실행 성능을 높이는 핵심 기법 중 하나가 객체 전달을 ‘상수 객체에 대한 참조자’(reference-to-const)로 진행하는 것.

위 기법이 제대로 쓰려면 상수 상태로 전달된 객체를 조작할 수 있는 const 멤버 함수, 즉 상수 멤버 함수가 준비되어 있어야 한다 것.

class TextBlock{

public:

const char& operator[](std::size\_t position) const // 상수 객체에 대한 operator[]

{return text[position];}

char& operator[] (std::size\_t position) // 비상수 객체에 대한 operator[]

{return text[position];}

private:

std::string text;

};

위 처럼 선언된 TextBlock의 operator[]는 다음과 같이 쓸 수 있습니다.

TextBlock tb(“Hello”); // TextBlock::operator[]의 비상수 멤버를 호출합니다.

std::cout << tb[0];

const TextBlock ctb(“world”); // TextBlock::operator[]의

std::cout << ctb[0]; // 상수 멤버를 호출합니다.

실제 프로그램에서 상수 객체가 생기는 경우는 1. 상수 객체에 대한 포인터 / 2. 상수 객체에 대한 참조자로 객체가 전달될 경우 이다.

아래의 예제가 더 실제의 경우와 가깝다.

void print(const TextBlock& ctb) // 이 함수에서 ctb는 상수 객체로 쓰입니다.

{

std::cout << ctb[0]; // TextBlock::operator[]의 상수

… // 멤버를 호출 합니다.

}

operator[]를 ‘오버로드(overload)해서 각 버전마다 반환 타입을 다르게 가져갔기 때문에, TextBlock의 상수 객체와 비상수 객체의 쓰임새가 달라짐

std::cout << tb[0]; // 좋습니다. 비상수 버전의 TextBlock객체를 읽습니다.

tb[0] = ‘x’; // 역시 문제없죠. 비상수 버전의 TextBlock 객체를 씁니다.

std::cout << ctb[0]; // 이것도 됩니다. 상수 버전의 TextBlock 객체를 읽습니다.

ctb[0] = ‘x’; // 에러! 상수 버전의 TextBlock 객체에 대해 쓰기는 안 됨.

이 에러는 const char& 타입에 대입 연산을 시도했기 때문에 발생 한 것. 상수 멤버로 되어 있는 operator[]의 반환 타입이 const char&이니까

operator[]의 비상수 멤버는 char의 참조자(char&)를 반환한다는 것인데 char만 쓰면 안된다는것을 기억해야한다. 만약 char만 쓰게 되면 tb[0] = ‘x’; // 비상수 버전임에도 에러 발생

이렇게 되는 이유는 기본제공 타입을 반환하는 함수의 반환 값을 수정하는 일은 절대로 있을 수 없기 때문입니다. 즉, 수정되는 값은 tb.text[0]의 사본이지, tb.text[0]자체가 아님

어떤 멤버함수가 상수 멤버라는 것은 양대 개념이 자리 잡고 있음

1. 비트수준 상수성(물리적 상수성)

어떤 멤버 함수가 그 객체의 어떤 데이터 멤버도 건들리지 않아야(정적멤버 제외) 그 멤버 함수가 const임을 인정. 상수 멤버 함수는 호출된 객체의 어떤 비정적 멤버도 수정할 수 없게 되어 있음

class CTextBlock {

public:

…

char& operator[](std::size\_t position) const // 부적절한(비트수준 상수성은 허용)

{ return pText[position]; }

private:

char\* pText;

};

코드에 나와 있듯이 operator[] 함수가 상수 멤버 함수로 선언 되어 있음.(틀린 것임에도) 이 함수는 해당 객체의 내부 데이터에 대한 참조자를 버젓이 반환 합니다. 하지만 operator[]의 내부 코드만 보면 pText는 확실히 안 걸드립니다.

하지만 이로 인해 어떤 사태가 생길지는 아래에서 확인해 보도록 합시다.

const CTextBlock cctb(“Hello”); // 상수 객체를 선언합니다.

char\* pc = &cctb[0]; // 상수 버전의 operator[]를 호출하여 cctb의 내부 데이터에 대한 포인터를 얻음

\*pc = ‘J’; // cctb는 이제 “Jello”라는 값을 갖습니다.

어떤 값으로 초기화된 상수 객체를 하나 만들어 놓고 이것에다 상수 멤버 함수를 호출했더니 값이 변해버린 것.

1. 논리적 상수성

위 처럼 황당한 상황을 보완하는 대체 개념으로 나오게 되었음. 상수 멤버 함수라고 해서 객체의 한 비트도 수정할 수 없는 것이 아니라 일부 몇 비트 정도는 바꿀 수 있되, 그것을 사용자측에서 알아채지 못하게만 하면 상수 멤버 자격이 있다는 것. 예를 들어 CTextBlock 클래스는 뭄장 구역의 길이를 사용자들이 요구할 때마다 이 정보를 캐시해 둘 수 있다.

예)

class CTextBlock {

public:

…

std:: size\_t length() const;

private:

char\* pText;

std::size\_t textlength; // 바로 직전에 계산한 텍스트 길이

bool lengthIsValid; // 이 길이가 현재 유효한가?

};

std::size\_t CTextBlock::length() const

{

if ( ! lengthIsValid) {

textLength = std::strlen(pText); // 에러! 상수 멤버 함수 안에서는

lengthIsValid = true; // textLength 및 lengthIsValid에 대입 할 수 없습니다.

}

return textLength;

}

length의 구현은 너무나도 자명하게 ‘비트수준 상수성’과 멀리 떨어져 있습니다. textLength 및 lengthIsValid가 바뀔 수 있으니까. 그렇지만 CTextBlock의 상수 객체에 대해서는 당연히 아무 문제가 없어야 할 것 같은 코드입니다. 컴파일러는 에러를 쏟아낼 게 뻔합니다. 비트 수준의 상수성이 지켜져야 합니다. 이런 상황에서 해답은 mutable을 사용하는 것입니다. mutable은 비정적 데이터 멤버를 비트수준 상수성의 족쇄에서 풀어 줍니다.

class CTextBlock {

public:

…

std::size\_t length() const;

private:

char\* pText;

mutable std::size\_t textlength; // 이 데이터 멤버들은 어떤 순간에도 수정이

mutable bool lengthIsValid; // 가능합니다. 심지어 상수 멤버 함수 안에서도 수정할 수 있습니다.

};

std::size\_t CTextBlock::length() const // 상수함수

{

if ( ! lengthIsValid) {

textLength = std::strlen(pText); // 이제 문제없습니다. lengthIsValid = true; // 당연히 문제없죠.

}

return textLength;

}

결론:

const 객체는 모든 멤버변수가 const가 된다.

const함수를 호출 하려면 상수객체가 멤버함수를 호출하면 됨.

그리고 또 const함수를 호출하려면 void print(const TextBlock& ctb){ std::cout << ctb[0];}

const객체임에도 모든 멤버변수가 const가 되기 싫으면 mutable을 붙이면 그것만 값을 수정 할 수 있다.

상수 멤버 및 비상수 멤버 함수에서 코드 중복 현상을 피하는 방법

mutable은 괜찮은 방법이지만 골칫거리 전부를 씻어낼 수는 없다.

예를 들어 TextBlock의 operator[] 함수가 지금은 특정 문자의 참조자만 반환하고 있지만, 이것 말고도 여러 가지를 더 할 수도 있을 것입니다. 이럴 경우 어느덧 코드 판박이(const함수냐 아니냐만 다를뿐 내용은 같은 코드) 괴물이 나뒹굴고 있게 됩니다.

코드 중복이 많아 지면 컴파일 시간, 유지보수, 복잡함 등 여러가지 문제를 일으킬 수 있습니다.

이 문제를 해결하려면 const 껍데기를 캐스팅으로 날리면 됩니다.

기본적으로 캐스팅은 일반적으로도 통념적으로도 썩 좋지 않은 아이디어지만 코드 중복보단 낫다.

const키워드가 붙는것을 빼면 완전히 같은 함수 이므로 캐스팅을 사용하여도 안전합니다.

그래서 결론은 안전성도 유지마면서 코드 중복을 피하는 방법은 비상수 operator[]가 상수 버전을 호출하도록 구현하는 것입니다.

예)

class TextBlock{

public:

...

const char& operator[](std::size\_t position) const // 이전과 동일

{

...

return text[position];

}

char& operator[](std::size\_t position) // 상수 버전 op[]를 호출하고 끝

{

return const\_cast<char&>( static\_cast<const TextBlock&>(\*this)[position] );

/\*op[]의 반환 타입에 캐스팅을 적용, const를 떼어냅니다. \*this의 타입에 const를 붙입니다. op[]의 상수 버전을 호출합니다. \*/

};

}

...

};

위 코드를 보시면 캐스팅이 한번이 아니라 두번 되어 있습니다.

자, 지금 해야 하는 일은 비상수 operator[]가 상수 버전을 호출하게 하는 것입니다. 그런데 비상수 operator[]안에서 그냥 operator[]라고 적으면 그 자신이 재귀적으로 호출되고 맙니다.

이런 재귀호출을 피하기 위해서는 상수 operator[]를 호출하고 싶다고 표현해주어야 합니다. 이 표현을 하기 위해 \*this의 타입 캐스팅입니다. 즉, 원래의 타입인 TextBlock&에서 const TextBlock&으로 바꾸는 것이죠.

결국 정리하면, 두 개의 캐스팅 중 첫 번째 것은 \*this에 const를 붙이는 캐스팅이고(비상수 operator[]에서 상수 버전을 호출하기 위해), 두 번째 것은 상수 operator[]의 반환 값에서 const를 떼어내는 캐스팅 입니다.

const를 붙이는 캐스팅은 안전한 타입 변환(비상수 객체에서 상수 객체로 바꾸는)을 강제로 진행하는 것뿐이기 때문에 static\_cast만 써도 딱 맞습니다. 반면에 const를 제거하는 캐스팅은 const\_cast밖에 없으므로 별다른 선택의 여지가 없습니다.

앞의 방법을 뒤집어서 하는 쪽(코드 중복 회피를 위해 상수 버전이 비상수 버전을 호출하게 만드는 것)도 생각할 수 있는데 이는 옳지 않다. 상수 멤버 함수는 해당 객체의 논리적인 상태를 바꾸지 않겠다고 컴파일러와 굳게 약속한 함수인 반면, 비상수 멤버 함수는 이런 약속 같은 걸 하지 않습니다.

즉, 어쩌다가 상수 멤버에서 비상수 멤버를 호출하게 되면, 수정하지 않겠다고 약속한 그 객체를 배신하는 셈이 되고 그 객체는 변경될 위험에 빠질 수 있습니다. 상수 멤버 함수에서 비상수 멤버 함수를 호출하려면 const\_cast를 적용해서 \*this에 붙은 const를 떼어내야 하는데, 이게 온갖 재앙의 씨앗입니다.

비상수 멤버 함수안에서는 그 객체를 바꾸든 안 바꾸든 맘대로 할 수 있기 때문에, 비상수 멤버함수에서 상수 멤버 함수를 호출한다고 해서 특별히 잘못될 리가 없습니다. 위의 코드에서 보셨듯이 \*this에 대해 static\_cast가 제대로 작동하는 이유가 바로 이 때문입니다. const에 관련된 위험이 없거든요.

정리

const를 붙여 선언하면 컴파일러가 사용상의 에러를 잡아내는 데 도움을 줍니다. const는 어떤 유효범위에 있는 객체에도 붙을 수 있으며, 함수 매개변수 및 반환 타입에도 붙을 수 있으며, 멤버 함수에도 붙을 수 있습니다.

컴파일러 쪽에서 보면 비트수준 상수성을 지켜야 하지만, 여러분은 개념적인 상수성을 사용해서 프로그래밍해야 합니다.

상수 멤버 및 비상수 멤버 함수가 기능적으로 서로 똑같게 구현되어 있을 경우에는 코드 중복을 피하는 것이 좋은데, 이때 비상수 번전이 상수 버전이 상수 버전을 호출하도록 만드세요.

**항목 4: 객체를 사용하기 전에 반드시 그 객체를 초기화하자**

생성자(): 여기서 하는 것이 초기화 이다. (복사생성자에 의해 초기화 됨)

{

//여기서 하는 것은 초기화가 아니다 “대입” 이다.

// 효율이 떨어진다. 이유는

// 초기화하는 부분에서 멤버변수에대해 기본 생성자를 호출하고 나서

// 바로 값을 대입하고 있다. 즉, 기본 생성자를 호출한 것이 헛짓이 되고 말았다.

}

상수이거나 참조자로 되어 있는 데이터 멤버의 경우엔 반드시 초기화 해야한다. 이유는 대입 자체가 불가능 하기 때문입니다.

결론은 멤버 초기화 리스트를 항상 사용하는 것이다.

초기화 순서는 1. 기본 클래스는 파생 클래스보다 먼저 초기화 됨.

2. 클래스 테이터 멤버는 선언된 순서대로 초기화 됨.

멤버 초기화 리스트에 넣는 멤버들의 순서도 클래스에 선언한 순서와 동일하게 맞춰 주자.

비지역 정적 객체의 초기화 순서는 개별 번역 단위에서 정해진다는 사실입니다.

정적 객체의 범주에 들어가는 것들

* 전역 객체가 있다. (=비지역 정적 객체)
* 네임스페이스 유효범위에서 정의된 객체(=비지역 정적 객체)
* 클래스 안에서 static으로 선언된 객체(=비지역 정적 객체)
* 함수 안에서 static으로 선언된 객체(=지역 정적 객체)
* 파일 유효범위에서 static으로 정의된 객체(=비지역 정적 객체)

이들 정적 객체는 프로그램이 끝날 때 자동 소멸됩니다.

번역 단위 = 컴파일을 통해 하나의 목적 파일을 만드는 바탕이 되는 소스 코드

여기서 번역은 소스의 언어를 기계어로 옮기는 것을 의미

#include하는 파일까지 합쳐서 하나의 번역 단위가 됩니다.

별도로 컴파일된 소스 파일이 두 개 이상 있으며 각 소스 파일에 비지역 정적 객체가 한 개 이상 들어 있는 경우에 어떻게 되느냐 하는 것. 그리고 실제적인 문제는 이것 -> 한쪽 번역 단위에 있는 비정적 객체의 초기화가 진행되면서 다른 쪽 번역 단위에 있는 비지역 정적 객체가 사용되는데, 불행히도 이(다른 쪽 번역 단위에 있는) 객체가 초기화되어 있지 않을지도 모른다는 점.

이러한 문제가 발생하는 것은 **별개의 번역 단위에서 정의된 비지역 정적 객체들의 추기화 순서는 ‘정해져 있지 않다’라는 사실 때문이다.**

**예제**)

class FileSystem { // 여러분의 라이브러리에 포함된 클래스

public:

std::size\_t numDisks() const; // 많고 많은 멤버 함수들 중 하나

};

extern FileSystem tfs; // 사용자가 쓰게 될 객체 “tfs” = “the file system”

파일 시스템 내의 디렉토리를 나타내는 클래스를 사용자가 많들었다고 가정

class Directory {

public:

Directory( params );

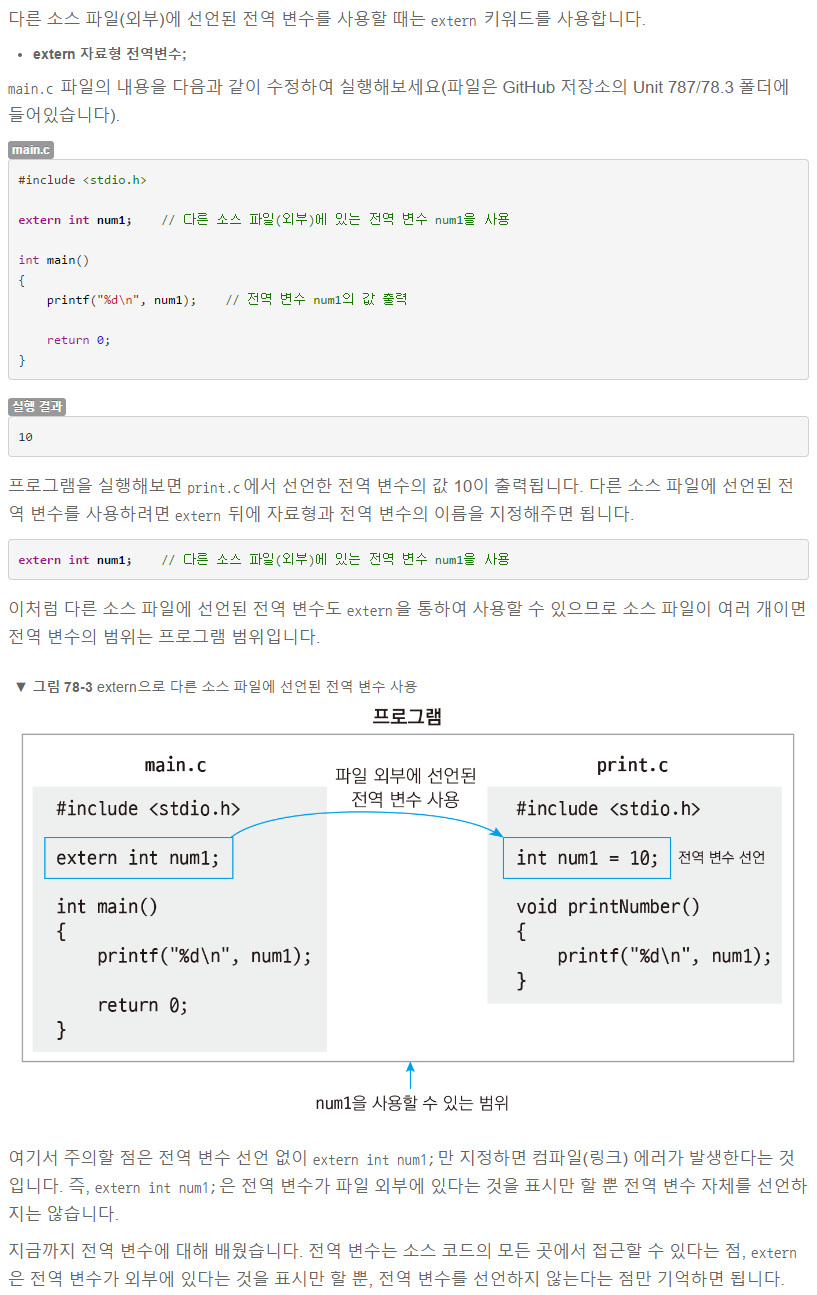
};

Directory::Directory( params )

{

std::size\_t disks = tfs.numDisks(); // tfs 객체를 여기서 사용함.

}



이제는 이 사용자가 Directory클래스를 사용해서 임시 파일을 담는 디렉토리 객체 하나를 생성하기로 마음먹습니다.

Directory tempDir( params ); // 임시 파일을 담는 디렉토리

이때 tfs가 tempDir보다 먼저 초기화되지 않으면, tempDir의 생성자는 tfs가 초기화 되지도 않았는데 tfs를 사용하려고 하겠지요. 그러나 tfs와 tempDir은 제작자도 다르고 만들어진 시기도 다른데다가 소재시(소스파일)도 다릅니다. 다시 말해 이들은, 다른 번역 단위 안에서 정의된 비지역 정적 객체입니다. 어쨌든 tempDir 전에 tfs가 초기화되게 만들고 싶은데, 어떻게 할 수 있을까?

단언컨대 안 됩니다. 서로 다른 번역 단위에 정의된 비지역 정적 객체를 사이의 상대적인 초기화 순서는 정해져 있지 않습니다.

설계에 약간의 변화만 주면 이 문제를 사전에 봉쇄할 수 있다는 점 입니다.

비지역 정적 객체를 하나씩 맡는 함수를 준비하고 이 안에 각 객체를 넣는 것입니다. 함수 속에서도 이들은 정적 객체로 선언하고, 그 함수에서는 이들에 대한 참조자를 반환하게 만듭니다. 사용자 쪽에서는 비지역 정적 객체를 직접 참조하는 과거의 폐단을 버리고 이제는 함수 호출로 대신합니다. 자, 정리하면 ‘비지역 정적 객체’가 ‘지역 정적 객체’로 바뀐 것입니다.

지역 정적 객체는 함수 호출 중에 그 객체의 정의에 최초로 닿았을 때 초기화되도록 만들어져 있습니다. 따라서 비지역 정적 객체를 직접 접근하지 않고 지역 정적 객체에 대한 참조자를 반환하는 쪽으로 바꾸었다면, 여러분이 얻어낸 참조자는 반드시 초기화된 객체를 참조하도록 맞추어 주어야 합니다.

‘지역 객체 참조자 반환’ 함수를 호출할 일이 없다면 해당 객체의 생성/소멸 비용도 생기지 않게 막아야 합니다.

class FileSystem{ ... }; // 이전과 다를 것 없는 클래스

FileSystem& tfs() // tfs 객체를 이 함수로 대신합니다. 이 함수는 클래스 안에 정적 { // 멤버로 들어가도 됩니다.

static FileSystem fs; // 지역 정적 객체를 정의하고 초기화합니다.

return fs; // 이 객체에 대한 참조자를 반환합니다.

}

class Directory { ... }; // 역시 이전과 다를 것 없는 클래스

Directory::Directory( params ) // 이전과 동일합니다. tfs의 참조자였던 것이

{ // 지금은 tfs()로 바뀌었다는 것만 다릅니다.

std::size\_t disks = tfs().numDisks();

}

Directory& tempDir() // tempDir 객체를 이 함수로 대신합니다. 이 함수는

{ // Directory 클래스의 정적 멤버로 들어가도 됩니다.

static Directory td; // 지역 정적 객체를 정의 / 초기화 합니다.

return td; // 이 객체에 대한 참조자를 반환합니다.

}

tfs와 tempDir대신에 tfs()와 tempDir()을 참조하는 것으로 바뀐게 전부이다. 즉, 정적 객체 자체를 직접 사용하지 않고 그 객체에 대한 참조자를 반환하는 함수를 사용하고 있는 것. 그러나 함수의 내부적으로 정적 객체를 쓰기 때문에, 다중스레드 시스템에서는 동작에 장애가 생길 수도 있다.

물론 초기화 순서 문제를 방지하기 위해 이처럼 참조자 반환 함수를 사용하는 아이디어는 여러분이 객체들의 초기화 순서를 제대로 맞춰 둔다는 전제조건이 뒷받침되어 있어야 한다.

정리

어떤 객체가 초기화 되기전에 그 객체를 사용하는 일이 생기지 않도록 하려면

첫째, 멤버가 아닌 기본제공 타입 객체는 여러분 손으로 직접 초기화 할 것.

둘째, 객체의 모든 부분에 대한 초기화에는 멤버 초기화 리스트를 사용합니다.

셋째, 별개의 번역 단위에 정의된 비지역 정적 객체에 영향을 끼치는 불확실한 초기화 순서를 염두에 두고 이러한 불확실성을 피해서 프로그램을 설계해야 합니다. -> 비지역 정적 객체를 지역 정적 객체로 바꾸기.

2. 생성자, 소멸자 및 대입 연산자

생성자 : 새로운 객체를 메모리에 만드는 데 필요한 과정을 제어

소멸자 : 객체를 없앰과 동시에 그 객체가 메모리에서 적절히 사라질 수 있도록 하는 과정을 제어

대입 연산자 : 기존의 객체에 다른 객체의 값을 줄 때 사용하는 함수이다.

**항목 5: C++가 은근슬쩍 만들어 호출해 버리는 함수들에 촉각을 세우자**

클래스 안에 직접 선언해 넣지 않으면 저절로 만들어지는 것은 기본 생성자, 복사 생성자, 복사 대입 연산자, 소멸자 이다. 기본형으로 만들어진다.

이때 소멸자는 이 클래스가 상속한 기본 클래스의 소멸자가 가상 소멸자로 되어 있지 않으면 역시 비가상 소멸자로 만들어진다는 점이다.

데이터 멤버가 참조자 또는 상수 객체 이고 ‘=’연산자를 통해 다른 객체의 값을 복사할 경우 기본으로 만들어지는 얕은 복사로는 컴파일 에러가 발생한다.

이유는 값을 복사 받는 쪽의 참조자가 한번에 2개 이상 참조할 수도 없고 원래 참조하던 곳을 버리고 복사 받는 쪽을 참조한다고 쳐도 다른 객체까지 영향을 받을 수 있기 때문이다.

또한 상수멤버의 경우 수정하는 것은 문법에 어긋나기 때문에 에러가 발생한다.

그러므로 멤버중에 상수 or 참조자가 있다면 꼭 복사 대입 연산자를 직접 만들어 주어야 한다.

**특별 항목: 포인터와 참조자를 구분하자**

null 참조라는 없기 때문에 어떤 경우든지 메모리 공간을 차지한 객체를 참조하고 있어야 합니다. 따라서 어떤 객체를 참조하는 변수를 만들 때 그 변수가 참조하는 부분에 꼭 객체가 있지 않을 경우도 있다고 한다면 포인터를 써야 합니다. 반대로 반드시 객체가 존재해야 한다면(null가능성 제로) 참조자를 써야 합니다.

char\* pc = nullptr; // 포인터를 null로 세팅합니다.

char& rc = \*pc; // nullptr을 역참조한 것을 참조자를 통해 참조하려고 함, 아주 괘씸하고 못된 코드

참조자는 반드시 객체를 참조하고 있어야 하기 때문에, C++ 스펙에 의하면 참조자는 선언될 때 반드시 초기화해야 합니다.

string& rs; // 에러! 참조는 반드시 초기화 해야 합니다.

string s(“xyzzy”);

string& rs = s; // ok! rs는 s를 참조합니다.

포인터를 쓸 경우에는 이런 제약이 없습니다.

string\* ps; // 초기화가 되지 않은 포인터 써도 상관없지만 문제가 생길 수 있습니다.

nullptr을 참조하는 참조자는 존재하지 않기 때문에 유효성 검사를 할 필요없다는 점에서 포인터보다는 참조자를 쓰는 것이 더 효율적일 수 있습니다.

포인터와 참조자가 다른 점 또 하나는 “다른 객체를 재참조하게 할 수 있는가”의 여부입니다.

포인터는 다른 객체의 주소값으로 얼마든지 바꾸어 세팅할 수 있습니다. 하지만 참조자는 초기화될 때 참조했던 그 객체만 참조합니다.

string s1(“Nancy”);

string s2(“Clancy”);

string& rs = s1; // rs는 s1을 “참조합니다.”

string\* ps = &s1; // ps는 s1을 “가리킵니다.”

rs = s2; // rs는 여전히 s1을 가리키지만, s1의 값은 이제 “Clancy”입니다. (값이 바뀐 것)

ps = &s2; // ps는 이제 s2를 가리킵니다. 그리고 s1은 바뀌지(값) 않습니다. (가리키는 대상이 바뀐 것)

일반적으로, 포인터를 써야 하는 경우는 딱히 가리킬 객체의 주소가 없을 때(즉, 포인터로는 nullptr로 초기화)이거나, 하나의 변수를 가지고 여러 개의 객체를 바꾸어 참조해야 할 때 입니다.

그리고 참조자를 써야 하는 경우는 참조할 포인터가 처음부터 끝까지 존재할 것임을 알고 있을 때와 참조하는 대상 객체를 바꿀 필요가 없을 때 입니다.

참조자를 반드시 써야 하는 특수한 상황이 있습니다. 바로 연산자 함수를 구현할 때 입니다.

예를 들면 operator[ ]인데, 이 연산자는 대입 연산자의 좌변(대입 대상)으로 쓸 수 있는 값을 반환해 주도록 만드는 것이 보통입니다.

vector<int> v(10); // 크기 10의 int 벡터를 만듭니다. 벡터는 표준 템플릿 라이브러리임.

v[5] = 10; // 이 대입 연산의 대상은 operator[ ]의 반환값입니다.

만일에 operator[ ]가 포인터를 반환하면, 앞의 마지막 문장은 다음과 같은 조금 어색한 형태가 되어야 할 것입니다.

\*v[5] = 10;

이 코드는 v가 포인터의 벡터인 것처럼 보이게 하는 단점을 가지고 있습니다. 실제로는 포인터의 벡터가 아닌데도 말이죠. 이 때문에 operator[ ]를 구현할 때에는 십중팔구 참조자를 반환하도록 해야 할 것입니다.

참조자는 참조하고자 하는 어떤 객체를 미리 알고 있을 때, 다른 객체를 바꾸어 참조할 일이 결코 없을 때, 그리고 포인터를 사용하면 문법상 의미가 어색해 지는 연산자를 구현할 때 선택하면 됩니다. 이 세 가지의 경우를 제외하고는 무조건 포인터입니다. 포!인!터!

**항목 6: 컴파일러가 만들어낸 함수가 필요 없으면 확실히 이들의 사용을 금해 버리자**

예를들어 복사를 금하는 객체가 있을 경우

복사 생성자 및 복사 대입 연산자를 private 멤버로 선언하도록 합시다. 여기까지가 90점짜리 정답이다.

이유 = private 멤버 함수는 그 클래스의 멤버 함수 및 프렌드 함수가 호출할 수 있다는 점이 여전히 허점입니다.

그럼 100점자리 정답은 무엇이냐? 바로 정의(구현)을 하지 않는 방법 입니다.

즉, private 멤버로 선언과 동시에 정의하지 않는 것입니다.

한 가지 더

복사 생성자와 복사 대입 연산자를 private로 선언하되, 이것을 HomeForSale(자신)에 넣지 말고 별도의 기본 클래스에 넣고 이것으로부터 HomeForSale을 파생시키는 것입니다.

그리고 그 별도의 기본 클래스는 복사 방지만 맡는다는 특별한 의미를 부여합니다.

예)

class Uncopyable{

protected: // 파생된 객체에 대해서

Uncopyable(){} //생성과 소멸을

~Uncopyable(){} // 허용합니다.

private:

Uncopyable(const Uncopyable&); // 하지만 복사는 방지합니다.

Uncopyable& operator=(const Uncopyable&);

};

복사를 막고 싶은 HomeForSale 객체는 이제 이렇게 바꿔 봅시다. Uncopyable로 부터 상속받게 하고 그냥 내버려 두는 것으로 끝입니다.

class HomeForSale: private Uncopyable { // 복사 생성자도, 복사 대입 연산자도 이제는 선언되지 않습니다.

...

};

항목7: 다형성을 가진 기본 클래스에서는 소멸자를 반드시 가상 소멸자로 선언하자

시간 기록을 유지하는 방법은 활용에 따라 무궁무진함 따라서 TimeKeeper 이름을 가진 클래스를 기본 클래스로 만들어 놓고 용도에 따라 이것을 파생시키도록 설계한다.

class TimeKeeper {

public:

TimeKeeper();

~TimeKeeper();

...

};

class AtomicClock: public TimeKeeper { ... };

이 클래스의 혜택을 받는 사용자들은 시간 정보에 접근하고 싶고(시간이 어떻게 계산되는지는 신경안씀) 그래서 팩토리 함수(파생 클래스 객체에 대한 기본 클래스 포인터를 반환하는 함수)를 만들어 두면 좋다.

TimeKeeper\* getTimeKeeper(); // TimeKeeper 에서 파생된 클래스를 통해 동적으로 할당된 객체의 포인터를 반환합니다.

이때 getTimeKeeper 함수에서 반환되는 객체는 힙에 있게 되므로, 결국 메모리 및 기타 자원의 누출을 막기 위해 해당 객체를 적절히 삭제 해야 합니다.

TimeKeeper\* ptk = getTimeKeeper(); // TimeKeeper 클래스 계통으로부터 동적으로 할당된 객체를 얻습니다.

... 이 객체를 사용합니다.

delete ptk; // 자원 누출을 막기 위해 해제(삭제) 합니다.

객체 삭제를 사용자에게 전가하는 것은 에러 발생에 노출될 소지가 있다.

**항목 7: 다형성을 가진 기본 클래스에서는 소멸자를 반드시 가상 소멸자로 선언하자**

가상 소멸자를 선언하는 것은 그 클래스에 가상 함수가 하나라도 들어 있는 경우에만 한정하세요.

가상 소멸자가 있는 부모를 상속받아야 한다.

가상 소멸자가 없는 클래스 => [string / vector / list / set / tr1::unordered\_map 등] 있다.

혹시 비가상 소멸자를 가진 표준 컨테이너 등의 클래스를 써서 쓸모 있는 나만의 클래스를 만들고 싶었다면 자제하세요.

파생클래스 소멸자가 먼저 호출된 다음 기반 클래스 소멸자가 호출 됨.

항목 8: 예외가 소멸자를 떠나지 못하도록 붙들여 놓자(소멸자에서 예외를 일으키지 않도록 하자)

사용자가 어떤 a객체에 대해 직접 close를 호출해야 a객체가 소멸한다면 a객체를 관리하는 클래스를 만드는 것이 좋다. 예를들어 a객체관리클래스의 소멸자에서 a객체 close를 호출하는 코드를 미리 심어두는 것이다. 그러면 a객체관리클래스 소멸자가 호출되면 자동으로 a객체도 같이 소멸되기 때문이다.

그런데 만약 a객체관리클래스의 소멸자에서 a객체를 소멸하는 중에 예외를 던진다면 사용자는 언제 터질지 모르는 시한폭탄을 갖고 있는것과 마찬가지 이고 (디버깅하기 힘들다..) 즉, 사용자에게 에러를 처리할 기회를 주지 못한다는 문제가 있다.

따라서 예외가 소멸자 밖에서 (다른 함수에서) 일어나도록 해야 한다.

예시 코드)

class DBConn {

public:

void close() // 사용자에게 에러를 처리할 기회를 줌(디버깅에서 이 함수를 호출한 부분이 오류 부분이 된{ db.close(); closed = true; } // 다.)

~DBConn()

{

if( ! closed )

{

try { db.close(); } catch ( … ) { close 호출이 실패했다는 로그를 작성합니다. … }

}

}

private:

DBConnection db;

bool closed;

};

===================== 17년 4월 13일 ========================

항목 9 객체 생성 및 소멸 과정 중에는 절대로 가상 함수를 호출하지 말자

기본 클래스의 생성자가 호출될 동안에는, 가상 함수는 절대로 파생 클래스 쪽으로 내려가지 않습니다.

(파생 클래스가 만들어지지 않았는데 가상 함수를 호출해봐야 소용업습니닷…)

파생클래스 객체의 기본 클래스 부분이 생성되는 동안은, 그 객체의 타입은 바로 기본 클래스 입니다. 호출되는 가상 함수는 모두 기본 클래스의 것으로 결정됨. 이유 –> 파생클래스는 아직 초기화된 상태가 아니기 때문에, 아예 없었던 것처럼 취급하는 편이 최고로 안전하다는 것.

**해결법**

생성 및 소멸 여러가지 객체에 따른 각각 다른 함수를 호출하고자 할 때 일단 가상 함수로 만들어 생성자 및 소멸자에서 호출할 생각 말고 비가상 함수로 바꾸고 나서 파생 클래스의 생성자들로 하여금 필요한 로그 정보를 부모 생성자로 넘겨야 한다는 규칙을 만듭니다.

class Transaction(부모클래스) {

public :

explicit Transaction(const std::string& logInfo);

void logTransaction(const std::string& logInfo) const;

};

Transaction::Transaction(const std::string& logInfo)

{

…….

logTransaction(logInfo); // 비가상 함수를 호출합니다.

}

class BuyTransaction(자식클래스): public Transaction {

public:

BuyTransaction(parameters) : // 자식 클래스 생성자

Transaction(createLogString(parameters)){ … } // <- 부모클래스 생성자 호출

…

private:

static std::string createLogString(parameters);

};

기본 클래스 부분이 생성될 때는 가상 함수를 호출한다 해도 기본 클래스의 울타리를 넘어 내려갈 수 없기 때문에, 필요한 초기화 정보를 파생 클래스 쪽에서 기본 클래스 생성자로 올려주도록 만듦으로서 부족한 부분을 역으로 채울 수 있다는 것입니다.

17년 5월 15일

자원 관리에 객체를 쓰자 강조하고 싶은 것.

자원 해제를 일일이 직접 하다 보면 (delete를 쓰는 것) 언젠가 잘못을 저지르고 만다는 이야기

shared\_ptr에 객체를 넘길 때 배열은 사용하지 말 것 ->이유 소멸자 내부에서 delete하는 것이지 delete[] 하는게 아니다.

자원 누출을 막기 위해, 생성자 안에서 자원을 획득하고 소멸자에서 그것을 해제하는 RAII객체를 사용.

**항목 14: 자원 관리 클래스의 복사 동작에 대해 진지하게 고찰 하자**

**RAII클래스에 대해서는 반드시 복사가 되지 않도록 막아야 합니다.** 복사 함수를 private멤버로 만드는 것

class Uncopyable()

{

protected: 생성자, 소멸자

private: 복사생성자; 복사대입연산자;

}

class Lock: private Uncopyable

{

…

}

**관리하고 있는 자원에 대해 참조 카운팅을 수행합니다**.

자원을 사용하는 객체가 있는데 그 자원을 저 세상으로 보내면 안된다.

내가 만든 RAII클래스에 참조 카운팅 방식의 복사 동작을 넣고 싶을 때 shared\_ptr을 데이터 멤버로 넣으면, 간단히 해결이 가능하다. Lock을 예로 들면

private: Mutex\* mutexPtr; 이것을 shared\_ptr<Mutex> mutexPtr 이렇게 바꾸는 것. 다만 문제는 shard\_ptr은 카운트가 0이 되면 삭제가 되는 것. (우리는 삭제까지 바라지는 않음)

class Lock{

public:

explicit Lock(Mutex \*pm) // shared\_ptr을 초기화하는데, 가리킬 포인터

:mutexPtr(pm, unlock) // Mutex 객체의 포인터를 사용하고 삭제자로 unlock 함수를 사용합니다.

{

lock(mutexPtr.get()); // “get” 항목 15에서 확인

}

private:

shared\_ptr<Mutex> mutexPtr; // 원시 포인터 대신에

}; // 소멸자를 선언하지 않았다 ( “객체 소멸 과정을 잊은 게 아니라 컴파일러가 생성한 소멸자를 통해 동작한다” )

클래스의 소멸자는 비정적 데이터 멤버의 소멸자를 자동으로 호출하게 되어 있다.

**관리하고 있는 자원을 진짜로 복사합니다.**깊은 복사 할 것.

이것만은 ㄱ하자.

RAII 클래스에 구현하는 일반적인 복사 동작은 복사를 금지하거나 참조 카운팅을 해야 한다.

**항목 15: 자원 관리 클래스에서 관리되는 자원은 외부에서 접근할 수 있도록 하자**

스마트 포인터에서 .get()함수를 쓰는 것이 귀찮으면 operator 클래스() const { return 객체; }

하지만 암시적 변환으로 인한 오류가 발생할 수 있다.

“맞게 쓰기에는 쉽게, 틀리게 쓰기에는 어렵게” 만들어야 할 것.

실제 자원을 직접 접근해야 하는 기존 API들도 많기 때문에, RAII클래스를 만들 때는 그 클래스가 관리하는 자원을 얻을 수 있는 방법을 열어 주어야 합니다.

자원 접근은 명시적 변환 혹은 암시적 변환을 통해 가능합니다. 안전성만 따지면 명시적 변환이 대체적으로 더 낫지만, 고객 편의성을 놓고 보면 암시적 변환이 괜찮습니다.

항목 16 new 및 delete를 사용할 때는 형태를 반드시 맞추자

string\* stringArray = new string[100];

위의 코드로는 소멸자가 99번 불릴 턱이 없기 때문에 잘 못된 코드이다.

new연산자를 사용하게 되면 두 가지의 내부 동작이 발생하게 되는데 일단 메모리가 할당됩니다.(operator new라는 이름의 함수가 쓰인다.) 그 다음, 할당된 메모리에 대해 한 개 이상의 생성자가 호출됩니다.

delete 표현식을 쓸 경우에도 두 가지 내부 동작이 진행 되는데, 우선 기존에 할당된 메모리에 대해 한 개 이상의 소멸자가 호출되고, 그 후에 그 메모리가 해제 됩니다.

delete 연산자가 적용되는 객체는 몇 개 일까? -> 삭제되는 포인터는 객체 하나만 가리킬까? 객체 배열을 가리킬까?

단일 객체의 메모리 배치구조와 객체 배열에 대한 메모리 배치구조는 다르다. 배열을 만드는 힙 메모리에는 배열원소의 개수가 박혀 들어감. 때문에, delete 연산자는 소멸자가 몇 번 호출될지 쉽게 알 수 있음.(단일 객체용 힙 메모리는 이런 정보가 없음)

delete에 []을 붙여주면 delete는 포인터가 배열을 가리키고 있구나 라고 가정하게 됩니다. 그렇지 않을 경우 그냥 단일 객체라고 간주합니다.

string\* stringPtr1 = new string; // delete stringPtr1; // 객체 한 개를 삭제합니다.

string\* stringPtr2 = new string[100]; // delete[] stringPtr2; // 객체의 배열을 삭제합니다.

만약 stringPtr1에 "[]"을 붙이면 어떤일이 발생할까?

우선 delete는 앞쪽의 메모리 몇 바이트를 읽고 이것을 배열 크기라고 해석할 것. 그 크기 만큼 소멸자를 호출할테고 결국 그 메모리는 자신이 소멸시키려는 타입의 객체가 아니라는 사실을 깨닫게 될 것.

반대로 stringPtr2에 “[]”을 붙이지 않으면 어떤 일이 생길까? 소멸자 호출 횟수가 적어서 문제가 발생할 것.

결론 : new 표현식에 []를 썼으면 delete할때도 “[]”을 쓰고 반대로 new 표현식에 “[]”을 쓰지 않았다면 delete할 때 “[]”을 쓰지 않으면 됩니다.

동적 할당된 변수를 갖고있는 클래스가 생성자가 여러 개일 경우에 특히 이 규칙을 깊이 간직하길 바란다. 이유는 포인터 멤버를 초기화하는 부분인 생성자가 여러 개 이면 new 형태를 똑같이 맞출 수 밖에 없기 때문이다. (어떤 생성자에서 []을 쓰면 다른 모든 생성자에서도 []을 써야함) 그래야 delete에서 형태를 똑같이 맞출 수 있음

typedef string AddressLines[4];

배열 타입을 typedef타입으로 만들지 않는 것이 좋다. 되도록이면 c++라이브러리를 이용하자(string / vector 등)

항목 17: new로 생성한 객체를 스마트 포인터에 저장하는 코드는 별도의 한 문장으로 만들자

반환값 함수이름(스마트포인터, int) <- 이런 함수를 호출할 때

함수이름(shared\_ptr<Widget>(new Widget), priority()) 이렇게 해서 함수를 호출하게 되면 자원 누출 가능성이 있다. 이유는 함수 호출하기 전에 인자값을 만드는 과정에서 new Widget하여 자원을 쓰고 priority()함수를 호출했는데 오류가 발생하여 결국 shared\_ptr<Widget>()이 호출 되지 못하면 new Widget한 부분이 자원 누출로 이루어 질 수 있다.

이런 문제를 예방하려면 함수를 호출할 때 인자안에서 스마트 포인터를 만들지 말고 외부에서 만든 뒤(자원관리 객체에게 넘긴 뒤) 스마트 포인터를 넘겨야 한다.

shared\_ptr<Widget> pw(new Widget); // 이렇게 스마트 포인터를 만들고

함수이름(pw, priority()) // 호출한다.